AFCRL-62-19

USE OF A LIST-PROCESSING LANGUAGE IN PROGRAMMING SIMPLIFICATION PROCEDURES

S. R. PETRICK

JANUARY 1962

ELECTRONICS RESEARCH DIRECTORATE
AIR FORCE CAMBRIDGE RESEARCH LABORATORIES
OFFICE OF AEROSPACE RESEARCH
UNITED STATES AIR FORCE
BEDFORD MASSACHUSETTS

Requests for additional copies by Agencies of the Department of Defense, their contractors, and other government agencies should be directed to the:

Armed Services Technical Information Agency
Arlington Hall Station
Arlington 12, Virginia

Department of Defense contractors must be established for ASTIA services, or have their 'need-to-know' certified by the cognizant military agency of their project or contract.

All other persons and organizations should apply to the:

U. S. DEPARTMENT OF COMMERCE
OFFICE OF TECHNICAL SERVICES,
WASHINGTON 25, D. C.

AFCRL -62-19

USE OF A LIST-PROCESSING LANGUAGE IN PROGRAMMING SIMPLIFICATION PROCEDURES

S. R. PETRICK

PROJECT 4641
TASK 464102

JANUARY 1962

COMPUTER AND MATHEMATICAL SCIENCES LABORATORY
ELECTRONICS RESEARCH DIRECTORATE
AIR FORCE CAMBRIDGE RESEARCH LABORATORIES
OFFICE OF AEROSPACE RESEARCH
UNITED STATES AIR FORCE
BEDFORD MASSACHUSETTS

ABSTRACT

A list-processing computer programming language is valuable for the coding of such logical algorithms as arise in truth function simplification because of ease of coding, improved use of computer storage, and a reduction of limitations on the number of variables that can be handled. The effectiveness of an algorithm implemented by means of a list-processing system is much less dependent upon the characteristics of the particular computer used than if basic machine language instructions are employed. In combination with the ease of coding, this independence permits the evaluation of the relative effectiveness of several diverse algorithms or heuristic implementations of a single algorithm. This paper demonstrates the usefulness of the MIT 709 LISP I System in coding several procedures for determination of the prime implicants and irredundant forms of a truth function. The capabilities and limitations of these programs are discussed as is their application to evaluating the algorithms they represent.

CONTENTS

USE OF A LIST-PROCESSING LANGUAGE IN PROGRAMMING SIMPLIFICATION PROCEDURES

INTRODUCTION

This paper is primarily concerned with an analysis of the advantages of a list-processing language in programming switching circuit minimization algorithms. The MIT LISP I System was used in this study, and it can be considered typical of several contemplated or existing systems in drawing conclusions about the usefulness of a list-processing language in the simplification of truth functions.

The minimization problem has received considerable attention in recent years. Aside from a few isolated results, however, only the problem of finding the simplest normal equivalent of a truth function has been completely solved. In this problem, many writers have shown that only certain conjunctions of literals need be considered as possible clauses. Quine[1] calls such clauses 'prime implicants.' Simplification can then be considered in two parts: determination of the prime implicants (or perhaps a particular subset of them), and selection of various prime implicants whose alternation is a minimal normal form.

With respect to the first problem, a hasty perusal of the literature discloses a plethora of alternate treatments. Closer scrutiny, however (if we are patient enough to plow through the morass of different notations and definitions the authors have used so effectively to make our comparison more difficult) reveals that there are merely a large number of variations on a relatively small number of distinct themes. Whether the notation is that of the algebraic topologist or that of the logician, the differences

Received for publication December 1961.

are inherent in the methods themselves. These fundamentally different themes include:

1) Quine's[1] determination of the set of prime implicants from the developed normal (canonical or standard sum) form. Briefly, on an n-dimensional cube model (Quine, of course, used the notation of the logician) all pairs of (m-1)-dimensional cells are compared to determine whether or not they constitute an m-dimensional cell; the process is then repeated on the set of m-dimensional cells so produced.

2) Urbano and Mueller's[2] determination of a set of prime implicants (not necessarily the entire set) also from the developed normal form. In this approach all basic cells that emanate from a vertex are found. In Quine's terminology, all prime implicants that are subsumed by a given clause of the developed normal form are computed directly.

3) Quine's[3] determination of the prime implicants from any normal formula by iterated consensus taking. A discussion of this approach will be deferred until later.

The second problem, that of selecting prime implicants to make up minimal forms, is treated in different fashion by a number of researchers including Gazale,[4] Mott,[5] McCluskey,[6] and this writer.[7, 8] Quine[1, 3, 9] also spends many pages upon this problem, but aside from a few results concerning indispensable and absolutely dispensable prime implicants, he has little to say about the problem of selection.

PROGRAMMING TRUTH FUNCTION SIMPLIFICATION

The theoretical work on truth function simplification has been complemented by the coding of various of these methods for digital computers. Most of these programs are either unpublished or else merely referred to briefly in a theoretical paper as a means of effecting the methods of that paper. Prather[10] has

published an account of a numerical method of effecting Urbano and Mueller's[2] algorithms, and programs to mechanize various of the methods of the previous section have been coded at MIT, Columbia University, IBM, Remington Rand Univac, and the General Electric Company, to name a few. Many of these programs were written as tools for the actual simplification of truth functions as required by some application, such as the design of logical components for a digital computer.

These programs have much in common. Because they are of a logical rather than a computational nature, they are all written either in basic machine code or in a language not far removed from it. They characteristically require large amounts of storage for many lists of variable and changing lengths and a fair amount of storage for the instructions. Upwards of 2000 instructions are the general rule for a complete simplification procedure. These instructions are usually sufficiently complicated to require a tangled bird's nest of a flow chart even for those programmers who do not believe in flow charts.

Another common feature shared by all such programs is their limitation to some fixed maximum number of literals that can be handled. This number seldom exceeds ten, and even then one or another of the temporary storage blocks often turn out to be insufficient. The individual digits or bits of a computer word are frequently used to denote specific literals, precluding extension beyond the number of bits or digits in a word unless extensive coding changes are made. Such an arrangement, however, permits the parallel processing of the literals of a pair of clauses, usually by clever coding. This is useful in combining clauses, testing to see if one clause subsumes another, forming the consensus of two clauses, and so on.

It is obvious that the use of a list-processing system such as the MIT LISP System or IPL of Newell, Shaw, and Simon has

many advantages over the type of program just discussed. First, the coding is more quickly, compactly, and correctly written. Second, there is no necessity to reserve specific blocks of temporary storage for diverse purposes, running out of available storage for different reasons from problem to problem with unused storage still available in every case. Third, there is no limitation on the maximum number of variables a single program can handle. The only limitations are the upper limit on total storage available and the time limit that can be tolerated for the simplification of a truth function. The serial processing of literals within a clause alone increases the running time considerably, but for many applications this should prove of secondary importance to the several advantages of the system.

DESCRIPTION OF SIMPLIFICATION PROCEDURES CONSIDERED

To demonstrate the advantages LISP I offers, a selected set of the procedures mentioned in the introductory section were coded. Under the category of prime implicant determination, method (1) of Quine was coded in preference to method (2) of Urbano. Aside from the fact that the latter method is more difficult to program (it is more difficult to describe precisely in English as well), there is still another compelling reason in support of this choice. Quine's method sequentially scans various unordered lists, whereas Urbano's requires numerous decisions as to whether a particular clause is included among those of the developed normal form. Although this could be accomplished by a sequential scan, the situation fairly cries out for a table lookup.

Returning to Quine's method, a selected passage from Caldwell[11] will be quoted to describe the manner in which it is applied:

"In this method we start with the function expressed as a standard sum. The first step is to compare each term in the function with all other terms and apply the theorem $XY + XY' = X$

wherever possible. All terms which have thus combined to form shorter terms should be checked off, so that in writing the final transmission function we can be certain that every term in the standard sum is included.

"The process is repeated, comparing every pair of shorter terms, until no more combinations can be made. At the conclusion of this process there will be a group of terms which have not been checked off. These are known as the prime implicants of the function and we seek to write the final transmission function as a sum of terms taken from the prime implicants, but to reject those that are redundant."

The next problem coded in LISP I was method (3), beginning from any alternation of clauses and applying the following two rules:

"a) If one of the clauses of alternation subsumes another, drop the subsuming clause. (A clause A subsumes a clause B if all the literals of B are included among the literals of A.)

"b) Adjoin, as an additional clause of alternation, the consensus of two clauses. The conjunction AB with any duplicate literals deleted is called the consensus of xA and x'B, provided that it contains no letter both affirmed and negated.

"The operation (b) is to be regarded as not applying in case the consensus subsumes a clause already present. The two operations are to be performed as long as possible. The method (a), in particular, is to be performed as much as possible before and after each performance of (b). When neither is applicable further, we have the alternation of all and only the prime implicants."

The final procedure coded was the production of all irredundant normal forms from the set of prime implicants. A normal form is irredundant if it has no superfluous clauses and its clauses have no superfluous literals, or, equivalently, if it is an alternation of prime implicants none of which is superfluous. The simplest normal form must be included among these irredundant forms. The first of Quine's methods, having already been coded, was used to find the required prime implicants, but any other method could have been used. The previously mentioned method of the author was used instead of one of the others largely because of personal bias. Basically, it constructs an auxiliary function that is a conjunction of alternations. There is one conjunct for

each clause of the developed normal form, and each conjunct is an alternation of literals denoting those prime implicants that are subsumed by that clause. If this auxiliary function is expanded to normal form and subsuming clauses are deleted, the literals of the remaining clauses denote those sets of prime implicants that make up each of the possible irredundant forms. To conserve time and storage, subsuming terms are deleted from partially expanded results. The expressions so obtained are then further expanded, and subsuming clauses are again deleted. This method is continued until all conjuncts have been so processed and an alternation of conjuncts, one denoting each irredundant form, have been produced. Several alternative sets of heuristics are available and will be explained.

DESCRIPTION OF THE LISP I SYSTEM

A brief description of the LISP I System is necessary to illustrate how easily and compactly LISP-coded programs may be written. All examples of functions coded will be selected from assorted truth function minimization programs.

In the ensuing paragraphs, much has been taken from the LISP I Programmer's Manual[13] to which the interested reader is directed. LISP I is a programming system for the IBM 709 (or 704) for computing with symbolic expressions. It has a central core based on a class of recursive functions of symbolic expressions.

One of the features of LISP I is the use of conditional expressions, devices for expressing the dependence of quantities on propositional quantities. A conditional expression has the form $[p_1 \rightarrow e_1, \ldots, p_n \rightarrow e_n]$ where the p's are propositional expressions and the e's are expressions of any kind. It can then be read, 'If p_1 then e_1, otherwise if p_2 then $e_2, \ldots,$ otherwise if p_n then e_n.' If all p's and e's are defined, the value of a conditional expression is the value of e corresponding to the first (ordered left to right) true proposition p.

For example,

$$|x| = [x < 0 \rightarrow -x;\ T \rightarrow x].$$

In recursive definition, a function is defined by means of formulas in which the defined function occurs. This is illustrated by the following example:

$$n! = [n = 0 \rightarrow 1;\ T \rightarrow n \times [n - 1]!\ ].$$

The class of symbolic expressions treated by the LISP System (S expressions) are formed by using the special characters

$$\begin{matrix} . \\ ) \\ (\end{matrix}$$

and an infinite set of distinguishable atomic symbols. Atomic symbols are defined to be strings of capital Latin letters and digits. S expressions include all atomic symbols and expressions of the form $(e_1 \cdot e_2)$ where e_1 and e_2 are themselves S expressions. A list $(m_1, m_2, \ldots, m_n)$ of arbitrary length can be represented in terms of S expressions as $(m_1\ (m_2 \cdot (m_n \cdot \text{NIL}) \ldots)))$. Here NIL is an atomic symbol used to terminate lists.

The LISP System also permits a class of functions of S expressions. For example, the predicate eq [x;y], which is defined if and only if either x or y is an atomic symbol, equals T if x and y are the same symbol, and otherwise equals F. Other useful functions include: atom [x] is a predicate that has the value of T or F, depending on whether x is an atomic symbol or not.

cons [x;y] is defined for any x and y

cons $[e_1; e_2] = (e_1 \cdot e_2)$

car [x] is defined if and only if x is not an atomic symbol

car $[(e_1; e_2)] = e_1$

cdr [x] is also defined if and only if x is not atomic

cdr $[(e_1 \cdot e_2)] = e_2$

equal [x;y] is a predicate that has the value T if x and y are the same S expression, and has the value F otherwise

null [x] = eq [x;NIL]

append [x;y] has as arguments two lists x and y; this function combines its two arguments into one new list.

In terms of functions already defined,

append [x;y] = [null[x] → y; T → cons [car[x]; append [cdr[x];y]]]

delete [x;n] removes one occurrence of x from the list n.

delete [x;n] = [null[n] → NIL; equal [x; car[n]] → cdr[n]; T → cons[car[n]; delete [x; cdr [n]]]]

subt [m;n] removes from list m one occurrence of each element of list n.

subt [m;n] = [null [n] → m; T → subt [delete [car [n]; m]; cdr [n]]]

union [m;n] = append [subt [m;n]; n].

TYPICAL USE OF LISP I IN SIMPLIFICATION

With the useful functions defined in the last section, it will be possible to detail the LISP coding necessary to obtain all prime implicants of a truth function from the developed normal form by Quine's method. Most readers will be familiar with this algorithm, so it should provide an estimate of coding effort necessary for other simplification processes.

The function primp [m] produces all prime implicants from the developed normal form m, a list of lists. It is a list of clauses of the developed normal form, and each of these lists is itself a list of literals of the truth function to be simplified. Each list denoting a clause of the developed normal form must always contain the literals or their primes in the same order. Any convenient atomic symbols may denote the literals and their primes (for example, A, B C, ... and AP, BP, CP, ..., for their primes) as long as a unique symbol is used for each. The only prohibited exceptions are FAIL and DASH.

Abbreviating the S expression $(m_1 \cdot (m_2 \cdot (m_n \cdot NIL)\ldots)))$ by the list notation $(m_1, m_2, \ldots, m_n)$, an example of a correct argument for

the function primp (m) is primp [((AP, BP, C), (AP, B, C), (A, BP, C), (A, B, CP), (A, B, C))]. Information of this kind, punched on cards, constitutes the input to the program. The output or value of the function primp (m) is best described by an example. For the previous argument it is ((A, B, DASH), (DASH, DASH, C)), denoting the two prime implicants AB and C.

One possible coding of the function primp [m] is the following:

```
primp [ m] = primp 1 [ m; scanned [ m] ]
primp 1 [ m;n] = primp 2 [ m; n; subt [ n; m]]
primp 2 [ m; n; o] = [ null [ o] → m; T → append [ subt [ m; n] ;
    primp [ o] ]]
scanned [ m] = [ null [ m] → NIL; T → union [ scan [ car [ m];
    cdr [ m]]; scanned [ cdr [ m]]]]
scan [ x;n] = [ null [ n] → NIL; T → union [ comp[ x;car[ n]];
    scan [ x; cdr [ n]]]]
comp [m;n] = comp 1 [ m;n; compare [ m;n]]
comp 1 [ m;n;o] = [ equal [ last [ o]; FAIL] → NIL; T → cons[ m;
    cons [ n; cons [ o; NIL ]]]]
last [ m] = [ null [ m] → NIL; null [ cdr [ m]] → car [ m];
    T → last [ cdr [ m]]]
compare [ m;n] = [ null [ n] → NIL; eq [ car [ n]; DASH] →
    [ eq [ car [ m]; DASH] → cons [ DASH; compare [ cdr [ m];
    cdr [ n]]]; T → cons [ FAIL; NIL]]; eq [ car [ m]; DASH] →
    cons [ FAIL; NIL] ; eq [ car [ m]; car [ n]] → cons [ car[ m];
    compare [ cdr [ m]; cdr [ n]]]; T → cons [ DASH;
    strike 2 [ cdr [ m]; cdr [ n]]]]
strike 2 [ m;n] = [ null [ n] → NIL; eq [ car [ n]; DASH] →
    [ eq [ car [ m] ; DASH] → cons [ DASH; strike 2 [ cdr [ m];
    cdr [ n]]]; T → cons [ FAIL; NIL]]; eq [ car [ m];
    DASH] → cons [ FAIL; NIL]; eq [ car [ m]; car [ n]] →
    cons [ car [ m]; strike 2 [ cdr [ m]; cdr [ n]]]; T →
    cons [ FAIL; NIL]].
```

Before these functions are explained in greater detail, a few comments should be made. Auxiliary functions such as primp 1, primp 2, and comp 1 could easily be avoided without augmenting the other functions materially, but binding the arguments in the manner indicated saves considerable recomputation time which the LISP I System would otherwise perform. It should also be noted that the functions compare and strike 2 are easily written and virtually identical even though they may look complicated at first inspection because they are much longer than most LISP functions.

If the functions used to define primp [m] are examined more closely, it can be seen that the argument of scanned [m] has the same form as that of primp [m]. It is a list of clauses and scanned [m] compares every pair of clauses in the manner to be detailed under comp [m;n] and forms the union of all such comparisons. Scan [x;n] has as arguments a single clause x and a list of clauses n. It forms the union of all lists comp [x;y] where y is one of the clauses of list n. Comp [m;n] computes compare [m;n]; if its last element is the atomic symbol FAIL, the value of comp [m;n] is NIL. Otherwise, comp [m;n] = a list consisting of m, n, and compare [m;n]. Compare [m;n] has two arguments, each designating a clause. If these two clauses differ only in one literal which occurs primed in one and unprimed in the other, compare [m;n] is the same as either m or n with the primed/unprimed literal replaced by the atomic symbol DASH. If the clauses differ in any other way, compare [m;n] is a list with the atomic symbol FAIL as the last element. Last [m] is the last element of list m.

For a more thorough understanding of the previous functions, begin with compare [m;n] and work backwards to primp [m]. Even a cursory knowledge of LISP indicates its relative ease in coding this method of prime implicant determination.

EFFECTIVENESS OF PROGRAMS PRODUCED

Two other programs were LISP coded in addition to the one detailed in the preceding section, Quine's prime implicant determination by iterated consensus taking, and irredundant form computation by the method of this author. The coding of these prodedures is included in the Appendix. Programs for the latter procedure and the one of the previous section have been debugged and used on several test cases. Debugging was quite easy, not only because very few mistakes were made, but also because it was so easy to check the validity of each function separately. This section treats the usage of these programs and includes information obtained from the test cases as to their effectiveness and limitations.

The nature of the input has already been specified for the function primp [m]. It is exactly the same for irredundant [m] which computes all irredundant forms from the developed normal form. During the computation of irredundant [m], primp [m] is evaluated. The value of irredundant [m] has the form illustrated by

irredundant [((AP, BP, CP), (AP, BP, C), (A, BP, C),
(A, B, CP), (A, B, C))] = (((AP, BP), (A, B), (BP, C)),
((AP, BP), (A, B), (A, C))).

The two irredundant forms represented are A'B' + AB + B'C and A'B' + AB + AC. Another function, irredundant 1 [m], has the same argument but strips off and displays the clauses that must appear in any irredundant form. Its output is illustrated by

irredundant 1 [((AP, BP, CP), (AP, BP, C), (A, BP, C),
(A, B, CP), (A, B, C))] = ((((AP, BP))), (((BP, C))),
((BP, C)), ((A, C))).

The test cases run, using either an IBM 709 or 7090, include the following:

primp [((AP, BP, C), (AP, B, C), (A, BP, C), (A, B, C),
(A, B, CP))] = ((A, B, DASH), (DASH, DASH, C))

primp [((PP, QP, RP, SP), (PP, QP, R, SP), (PP, QP, R, S), (PP, Q, R, SP), (PP, Q, R, S), (P, QP, RP, SP), (P, QP, RP, S), (P, Q, RP, SP), (P, QP, RP, S), (P, Q, R, SP), (P, Q, R, S))] = (PP, QP, DASH, SP), (DASH, QP, RP, SP), (PP, DASH, R, DASH), (DASH, Q, R, DASH), (P, DASH, RP, DASH), (P, Q, DASH, DASH))

For the same argument m as in the previous case,

irredundant [m] = (((P, Q), (P, RP), (PP, R), (PP, QP, SP)), ((P, RP), (Q, R), (PP, R), (PP, QP, SP)), ((P, Q), (P, RP), (PP, R), (QP, RP, SP)), ((P, RP), (Q, R), (PP, R), (QP, RP, SP)))

irredundant 1 [((AP, BP, CP), (AP, BP, C), (AP, B, CP), (A, BP, CP))] = ((((AP, BP))), (((AP, CP))), (((BP, CP))))

Two more functions treated by means of irredundant [m] were[11]

Σ(0, 1, 2, 4, 5, 8, 10, 12, 13, 16, 17, 18, 19, 21, 24, 25, 26, 27, 29)

and

Σ(2, 5, 12, 22, 24, 25, 27, 28, 29, 30, 31, 32, 35, 37, 39, 45, 47, 49, 54, 56, 57, 58, 60, 61, 62, 64, 65, 66, 67, 69, 72, 75, 76, 80, 81, 94, 97, 98, 105, 106, 108, 113, 118).

In all cases, prime implicant determination took five minutes or less, but irredundant form determination ranged from negligible time up to $2\frac{1}{2}$ hours without termination for the last case given. No provisions had been made to use the common simplification procedures involving indispensable prime implicants, and when these were added, computation time was reduced to less than two minutes for this case.

In general, however, irredundant form computation still took much longer than prime implicant determination. In order to study this more closely, several examples of the function irred [m] were run. The forms of its argument and value are illustrated by

irred [(((A), (B), (D)), ((A), (C), (D)), ((B), (D), (E)), ((A), (E)))] = ((E, A), (A, B), (E, C, B), (A, D), (E, D))

which is the list notation for

(A + B + D) (A+C+D) (B+D+E) (A+E) = AB + AD + AE + DE + BCE.

Two other examples run were

(R1 + R9) (R1 + R2 + R8) (R3 + R6 + R9) (R4 + R5 + R7)
(R2 + R3 + R8 + R9) R4 + R6 + R8) (R4 + R6 + R7)
(R1 + R5 + R6 + R7) (R3 + R5 + R9) (R2 + R3 + R5)

and

(B + F) (B+G+L+Q+V) (B+H+W+T) (D+J) (D+I+N+S+X)
(D+H+L+P) (C+H+M+R+W) (C+I+O) (C+G+K) (O+S+W)
(J+N+R+V) (P+V) (F+L+R+X) (K+Q+V)(F+G+H+I+J)
(X+L+M+N+O) (P+Q+R+S+T) (T+X).

The latter function came from a chess board covering problem, and its solution was of practical interest. The former problem taxed and the latter swamped the time limitations of the computer. Consequently, new heuristics were added. Originally, the last two factors were expanded and simplified, then the resulting factor was expanded with the third from the last original factor and simplified, and so on. The new procedure tried was to expand the original factors in pairs, simplifying the result of each such expansion, and to repeat this process on the resulting factors until the function was completely expanded. This permitted computation of the first case in less than two minutes, but the second case was still unfinished in fifteen minutes. It was shown that the addition of a set of heuristics to divide the problem into a set of simpler problems by means of the well known method of branching again permits the solution of this particular problem in a few minutes. It would not be hard to propose a still more complicated sample which would swamp the capabilities of even this program, however.

During the course of an ordinary machine run, the LISP System is duplicated on a scratch tape. It is, therefore, trivial to make extra tapes containing all of the functions of this report. Any group interested in obtaining such a tape for production or further research may do so simply by supplying a blank tape.

CONCLUSIONS

It is apparent that LISP is a convenient language in which to program truth function simplification procedures. This would have been even more striking if many of the auxiliary functions used to bind variables and prevent wasteful recomputations had not been introduced. It is also significant that LISP imposes no reservations on the use of tabular temporary storage to be reserved for diverse purposes nor on the maximum number of variables a single program can handle. Use of the functions included in this paper should make it easier to implement other simplification procedures.

A LISP-coded comparison of two dissimilar methods of attaining the same goal might prove a persuasive argument in support of one of these methods. If speed performances consistently and strongly favor one method for a large number of truth functions considered, it might be concluded not only that this method is preferable for LISP implementation, but also that it is probably to be preferred for hand computation. At least, the LISP programs seem to operate in a manner closer to the way in which a human applies simplification rules than do basic language coded programs, which are more dependent upon the idiosyncracies of particular machines.

Looking more critically, the LISP program can be thought of as applying a sequence of procedures not unlike those into which a human has conceptually classified his complex simplification procedure. To deduce anything about a method's applicability to hand computation from a corresponding LISP program, the individual time or effort requirements of the constituent components of each must be known.

Then by means of a few inequalities (since the basic operations cannot be expected to pose the same relative difficulty for both man and machine) it is possible to hypothesize about the effectiveness of a set of methods for hand computation on the basis of experimental machine results. Care must be taken, however, in generalizing such results to other machines.

It was not possible for the author to make any such comparisons of several basically dissimilar methods of truth function simplification. Several alternative heuristics were tried, however, for carrying out individual algorithms considered, resulting in a determination of marked superiority for certain procedures that did not seem intrinsically better than others. This was particularly true in irredundant form computation as is treated in the Appendix. It lends credence to the thesis that empirical results obtained using the LISP Programming System may be useful for evaluating the relative effectiveness of diverse simplification procedures.

APPENDIX

Analogously to the way the functions defining primp [m] were themselves defined in terms of more elementary functions, this Appendix will treat the functions irredundant [m] and primp 3 [m] which respectively produce irredundant forms and prime implicants by iterated consensus taking. Explanations of the roles played by each function and descriptions of the form of their arguments and values will not be included. These are, however, available on request.

The list structure of both the argument and value of the function irredundant [m] has previously been discussed. This function can be defined as follows:

```
irredundant [ m] = irred [ table [ m; primp [ m] ] ]
table [ m;n] = [ null [ m] → NIL; T → cons [ column [ car [ m];n];
    table [ cdr [ m]; n] ] ]
column [ x;n] = [ null [ n] → NIL; subsume 1 [ x; delete 1
    [ DASH; car [ n] ] ] → cons [ cons [ delete 1 [ DASH; car [ n] ];
    NIL];column [ x; cdr [ n] ] ]; T → column [ x; cdr [ n] ] ]
delete 1 [ x;n] = [ null [ n] → NIL; equal [ x; car [ n] ] →
    delete 1 [ x; cdr [ n] ]; T → cons [ car [ n]; delete 1 [ x;
    cdr [ n] ] ] ]
irred [ m] = [ null [ m] → NIL; T → mult [ car [ m]; irred [ cdr [ m] ] ] ]
mult [ m;n] = mult 2 [ mult 1 [ m; n] ]
mult 2 [ m] = subsume 3 [ m; m]
mult 1 [ m;n] = [ null [ n] → m; null [ m] → NIL; T → append
    [ expand [ car [ m] ; n]; mult 1 [ cdr [ m]; n] ] ]
subsume 3 [ m;n] = [ null [ n] → m; subsume 2 [ car [ n];
    delete [ car [ n]; m] ]; subsume 3 [ delete [ car [ n]; m];
    cdr [ n] ]; T → subsume 3 [ m; cdr [ n] ] ]
subsume 1 [ x;y] = null [ subt [ y;x] ]
subsume 2 [ x;m] = [ null [ m] → F; subsume 1 [ x; car [ m] ] → T;
    T → subsume 2 [ x; cdr [ m] ] ]
```

expand [x;n] = [null [n] → NIL; T → cons [union [car [n];
expand [x; cdr [n]]]]

Additional and modified heuristics, most of which were discussed in the main body of this paper, include:

(A) expand [x; n] = null [n] → NIL; subsume 2 [x;n] → cons [x;NIL];
T → cons [union [car [n]; x]; expand [x;cdr [n]]]]

This substitution combines some simplification into the expansion process.

(B) irred [m] = [null [cdr [m]] → car [m]; T → irred [irred 1[m]]]
irred 1 [m] = [null [m] → NIL; null [cdr [m]] → m; T →
cons [mult [car [m]; cadr [m]]; irred 1 [cddr [m]]]]

This change and addition replace sequential, factor-by-factor expansion and simplification by a pairwise parallel process.

(C) irredundant 1 [m] = strip [irred 3 [table [m; primp [m]]]]
strip [m] = [equal [length [car [m]]; 1.0] → cons [car [m];
strip [cdr [m]]]; T → irred [m]]
length [y] = [null [y] → 0.0; T → sum [length [cdr [y]]; 1.0]]
irred 3 [m] = irred 2 [m; m]
irred 2 [m;n] = [null [n] → m; equal [length [car [n]]; 1.0] →
cons [car [n]; irred 3 [delete 2[car [n]; m]]]; T →
irred 2 [m; cdr [n]]]
delete 2 [m;n] = [null [n] → NIL; contained [car [m]; car[n]] →
delete 2 [m; cdr [n]] ; T → contained [x; cdr [y]]]

These additions strip off all indispensable prime implicants and delete all factors that contain an indispensable prime implicant as a term.

Prime implicant computation by iterated consensus taking might be accomplished by a function primp 3 [m] whose argument is the same as for primp [m]. The following coding has not been debugged:

primp 3 [m] = primp 4 [subsume 5 [m]]
primp 4 [m] = primp 5 [m; generate [m]]
primp 5 [m;n] = [null [n] → m; T → primp 6 [cons [n;m]]]

```
primp 6 [ m] = primp 4 [ subsume 6 [ m] ]
subsume 5 [ m] = subsume 8 [ m;n]
subsume 8 [ m; n] = [ null [ n] → m; subsume 4 [ car [ n];
    delete[car [ n]; m] ] → subsume 8 [ delete [ car [ n]; m]
    cdr [ n] ]; T → subsume 8 [ m; cdr [ n] ] ]
subsume 6 [ m] = subsume 9 [ car [ m]; cdr [ m] ]
subsume 9 [ x;n] = [ null [ n] → NIL; subsume 7 [ car [ n];
    x] → subsume 9 [ x; cdr [ n] ]; T → cons [ car[ n];
    subsume 9 [ x; cdr [ n] ] ] ]
generate [ m] = gen [ m;m]
gen [ m;n] = gen 1 [ m;n; trygen [ car [ m]; cdr [ m]; n] ]
gen 1 [ m;n;o] = [ null [ m] → NIL; null [ o] → gen [ cdr [ m];
        n]; T → o]
trygen [ x;m;n] = trygen 1 [ x;m;n; consensus 1 [ x; car [ m] ] ]
trygen 1 [ x;m;n;y] = [ null [ m] → NIL; eq [ last [ y];
    FAIL] ∨ subsume 4 [ y;n] → trygen [ x; cdr [ m]; n];
    T → cons [ y; m] ]
subsume 4 [ x;m] = [ null [ m] → F; subsume 7 [ x; car [ m] ] → T;
    T → subsume 4 [ x; cdr [ m] ] ]
subsume 7 [ x;y] = [ null [ y] → T; eq [ car [ y]; DASH] ∨
    eq [ car[ y]; car [ x] ] → subsume 7 [ cdr [ x]; cdr [ y] ];
    T → F]
consensus 1 [ m;n] = [ null [ n] → NIL; eq [ car [ m];
    car [ n] ] ∨ eq [ car[ m]; DASH] → cons [ car [ n];
    consensus 1 [ cdr [ m]; cdr [ n] ] ]; eq [ car [ n]; DASH] →
    cons [ car [ m]; consensus 1 [ cdr [ m]; cdr [ n] ] ]; T →
    cons [ DASH; consensus 2 [ cdr [ m]; cdr [ n] ] ] ]
consensus 2 [ m;n] = [ null [ n] → NIL; eq [ car [ m]; car [ n]] ∨
    eq [ car[ m]; DASH] → cons [ car [ n]; consensus 2 [ cdr [ m];
    cdr [ n] ] ]; eq [ car [ n]; DASH] → cons [ car [ m];
    consensus 2 [ cdr [ m]; cdr [ n] ] ]; T → cons [ FAIL;NIL] ]
```

REFERENCES

1. W. V. QUINE, The Problem of Simplifying Truth Functions, Amer. Math. Monthly, 59:521-531, 1952.
2. R. H. URBANO and R. K. MUELLER, A Topological Method for the Determination of the Minimal Forms of a Boolean Function, IRE Trans. EC, Ec-5, pp 126-132, 1956.
3. W. V. QUINE, A Way to Simplify Truth Functions, Amer. Math. Monthly, 61:627-631, 1955.
4. M. J. GAZALE, Irredundant Disjunctive and Conjunctive Forms of a Boolean Function, IBM Journal of R&D, I:171-176, 1957.
5. T. H. MOTT, Determination of the Irredundant Normal Forms of a Truth Function by Iterated Consensus of the Prime Implicants, Proceedings of the ICIP, Butterworths, London, June 1959.
6. E. M. McCLUSKEY and I. B. PYNE, An Essay on Prime Implicant Tables, Tech. Rpt No. 1, Dept. of E. E. Dig. Systems Lab., Princeton University, October 1960.
7. S. R. PETRICK, On the Minimization of Boolean Functions, Proceedings of the ICIP, Butterworths, London, June 1959.
8. S. R. PETRICK, A Direct Determination of the Irredundant Forms of a Boolean Function From the Set of Prime Implicants, AFCRC-TR-56-110, Air Force Cambridge Research Center, April 1956.
9. W. V. QUINE, On Cores and Prime Implicants of Truth Functions, Amer. Math. Monthly, 66:755-760, 1959.
10. R. PRATHER, Computational Aids for Determining the Minimal Form of a Truth Function, J. Assoc. Comp. Mach., Vol. 7, No. 4, October 1960.
11. S. H. CALDWELL, Switching Circuits and Logical Design, John Wiley & Sons, 1958, pp 146-147.
12. E. M. McCLUSKEY, Minimization of Boolean Functions, Bell Sys. Tech. J., 35:1417-1444, 1956.
13. LISP I Programmer's Manual, Computation Center and Research Lab. of Electronics, MIT, March 1960.

UNCLASSIFIED

AF Cambridge Research Laboratories, Bedford, Mass. Electronics Research Directorate
USE OF A LIST-PROCESSING LANGUAGE IN PROGRAMMING SIMPLIFICATION PROCEDURES, by S.R. Petrick. January 1962. 19 pp.
AFCRL-62-19 Unclassified report

A list-processing computer programming language is valuable for the coding of such logical algorithms as arise in truth function simplification because of ease of coding, improved use of computer storage, and a reduction of limitations on the number of variables that can be handled. The effectiveness of an algorithm implemented by means of a list-processing system is much less dependent upon the characteristics of the particular computer used than if basic machine language instructions are employed. In combination with the ease of coding, this independence permits the evaluation of the relative effectiveness of several diverse algorithms or heuristic implementations of a single algorithm. This paper demonstrates (over)

1. Computers—Data processing systems programming
2. Electrical electronic circuits—Switching circuits

I. Petrick, S.R.

UNCLASSIFIED

AF Cambridge Research Laboratories, Bedford, Mass. Electronics Research Directorate
USE OF A LIST-PROCESSING LANGUAGE IN PROGRAMMING SIMPLIFICATION PROCEDURES, by S.R. Petrick. January 1962. 19 pp.
AFCRL-62-19 Unclassified report

A list-processing computer programming language is valuable for the coding of such logical algorithms as arise in truth function simplification because of ease of coding, improved use of computer storage, and a reduction of limitations on the number of variables that can be handled. The effectiveness of an algorithm implemented by means of a list-processing system is much less dependent upon the characteristics of the particular computer used than if basic machine language instructions are employed. In combination with the ease of coding, this independence permits the evaluation of the relative effectiveness of several diverse algorithms or heuristic implementations of a single algorithm. This paper demonstrates (over)

UNCLASSIFIED

UNCLASSIFIED

AF Cambridge Research Laboratories, Bedford, Mass. Electronics Research Directorate
USE OF A LIST-PROCESSING LANGUAGE IN PROGRAMMING SIMPLIFICATION PROCEDURES, by S.R. Petrick. January 1962. 19 pp.
AFCRL-62-19 Unclassified report

A list-processing computer programming language is valuable for the coding of such logical algorithms as arise in truth function simplification because of ease of coding, improved use of computer storage, and a reduction of limitations on the number of variables that can be handled. The effectiveness of an algorithm implemented by means of a list-processing system is much less dependent upon the characteristics of the particular computer used than if basic machine language instructions are employed. In combination with the ease of coding, this independence permits the evaluation of the relative effectiveness of several diverse algorithms or heuristic implementations of a single algorithm. This paper demonstrates (over)

1. Computers—Data processing systems programming
2. Electrical electronic circuits—Switching circuits

I. Petrick, S.R.

UNCLASSIFIED

AF Cambridge Research Laboratories, Bedford, Mass. Electronics Research Directorate
USE OF A LIST-PROCESSING LANGUAGE IN PROGRAMMING SIMPLIFICATION PROCEDURES, by S.R. Petrick. January 1962. 19 pp.
AFCRL-62-19 Unclassified report

A list-processing computer programming language is valuable for the coding of such logical algorithms as arise in truth function simplification because of ease of coding, improved use of computer storage, and a reduction of limitations on the number of variables that can be handled. The effectiveness of an algorithm implemented by means of a list-processing system is much less dependent upon the characteristics of the particular computer used than if basic machine language instructions are employed. In combination with the ease of coding, this independence permits the evaluation of the relative effectiveness of several diverse algorithms or heuristic implementations of a single algorithm. This paper demonstrates (over)

UNCLASSIFIED

the usefulness of the MIT 709 LISP I System in coding several procedures for determination of the prime implicants and irredundant forms of a truth function. The capabilities and limitations of these programs are discussed as is their application to evaluating the algorithms they represent.	UNCLASSIFIED UNCLASSIFIED	the usefulness of the MIT 709 LISP I System in coding several procedures for determination of the prime implicants and irredundant forms of a truth function. The capabilities and limitations of these programs are discussed as is their application to evaluating the algorithms they represent.	UNCLASSIFIED UNCLASSIFIED
the usefulness of the MIT 709 LISP I System in coding several procedures for determination of the prime implicants and irredundant forms of a truth function. The capabilities and limitations of these programs are discussed as is their application to evaluating the algorithms they represent.	UNCLASSIFIED UNCLASSIFIED	the usefulness of the MIT 709 LISP I System in coding several procedures for determination of the prime implicants and irredundant forms of a truth function. The capabilities and limitations of these programs are discussed as is their application to evaluating the algorithms they represent.	UNCLASSIFIED UNCLASSIFIED